Journal of Discrete Algorithms 16 (2012) 206–212

Contents lists available at SciVerse ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

A priority queue with the time-finger property ☆

Amr Elmasry [a,*,1], Arash Farzan [b], John Iacono [c,2]

[a] Computer Science Department, University of Copenhagen, Denmark
[b] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[c] Polytechnic Institute of New York University, Brooklyn, New York, USA

ARTICLE INFO

Article history:
Available online 8 May 2012

Keywords:
Data structures
Priority queues
Distribution-sensitive structures
Working-set bound
Splay trees

ABSTRACT

We present a priority queue that supports *insert* in worst-case constant time, and *delete-min*, *access-min*, *delete*, and *decrease* of an element x in worst-case $O(\log(\min\{w_x, q_x\}))$ time, where w_x (respectively, q_x) is the number of elements that were accessed after (respectively, before) the last access to x and are still in the priority queue at the time when the corresponding operation is performed. (An *access* to an element is accounted for by any priority-queue operation that involves this element.) Our priority queue then has both the working-set and the queueish properties; and, more strongly, it satisfies these properties in the worst-case sense. From the results in Iacono (2001) [11] and Elmasry et al. (2011) [7], our priority queue also satisfies the static-finger, static-optimality, and unified bounds. Moreover, we modify our priority queue to realize a new unifying property — the time-finger property — which encapsulates both the working-set and the queueish properties.

1. Introduction

Distribution-sensitive data structures are those structures for which the time bounds to perform operations vary depending on the sequence of operations performed [11]. These data structures typically perform as well as their distribution-insensitive counterparts on a random sequence of operations in the amortized sense; yet, when the sequence of operations follows some particular distribution (for example, having temporal or spatial locality), the distribution-sensitive data structures perform significantly better.

The quintessential distribution-sensitive data structure is the splay tree [14]. Splay trees seem to perform very efficiently over several natural sequences of operations, both theoretically [14] (asymptotically faster than $\Theta(\log n)$ search time on a set of n elements) and practically [17]. There still exists no single comprehensive distribution-sensitive analysis for splay trees. Instead, there are theorems and conjectures characterizing their distribution-sensitive properties; these include: static finger, static optimality, working set [14], sequential access [5,15], unified bound [14], dynamic finger [4], and unified conjecture [1]. We refer the reader to [1,14] for thorough definitions and discussions of these properties.

Consider a sufficiently long sequence of access operations $\mathcal{A} = \langle a_1, a_2, \ldots, a_m \rangle$ (sorted in chronological order) performed on a set of n elements $\{x_1, x_2, \ldots, x_n\}$. Let e_i be the index of the element accessed by operation a_i. For a data-structural

☆ A preliminary version of this paper was presented at the 22nd International Workshop on Combinatorial Algorithms held in Victoria, Canada, in June 2011 (Elmasry et al., 2011) [7].

* Corresponding author.
E-mail addresses: elmasry@diku.dk (A. Elmasry), afarzan@mpi-inf.mpg.de (A. Farzan), jiacono@poly.edu (J. Iacono).

[1] Supported by fellowships from Alexander von Humboldt and VELUX foundations.
[2] Research partially supported by NSF grants CCF-0430849, CCF-1018370, and an Alfred P. Sloan fellowship, and by MADALGO—Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation, Aarhus University.

1570-8667/$ – see front matter
http://dx.doi.org/10.1016/j.jda.2012.04.014

realization that achieves the distribution-sensitive bounds (even in the amortized sense), the length of such access sequence should satisfy $m = \Omega(n \log n)$.

The *static-finger property* [14] indicates that, for any fixed element of index f (the finger), the amortized time to perform a_i is $O(\log(d_{\mathcal{A}}(e_i, f)))$, where $d_{\mathcal{A}}(e_i, f)$ is the difference in order between x_{e_i} and x_f when the elements are sorted by value. More specifically, for an access sequence $\mathcal{A}$, the total access time for a structure with the static-finger property is $O(\sum_{i=1}^{m} \log(d_{\mathcal{A}}(e_i, f)))$.

The *static-optimality property* (entropy bound) [14] indicates that, for an access sequence $\mathcal{A}$, where the element x_{e_i} is accessed $h_{\mathcal{A}}(e_i)$ times, the total access time is $O(\sum_{i=1}^{m} \log(m/h_{\mathcal{A}}(e_i))) = O(\sum_{j=1}^{n} h_{\mathcal{A}}(j) \cdot \log(m/h_{\mathcal{A}}(j)))$.

The *working-set size* $w_{\mathcal{A}}(i)$, for an operation a_i in sequence $\mathcal{A}$, is defined as the number of distinct elements accessed since the last access to the element x_{e_i}, or from the beginning of the sequence if this is the first access to x_{e_i}. The *working-set property* [14] indicates that the total access time for a sequence $\mathcal{A}$ is $O(\sum_{i=1}^{m} \log(w_{\mathcal{A}}(i)))$. Informally, the working-set property implies that the elements that have been recently operated on are faster to operate on compared to the elements that have not been accessed in the recent past.

The *unified bound* [14] indicates an apparently, but not indeed [7], stronger property. The unified bound is the minimum per operation among the static-finger, static-optimality, and working-set bounds. More precisely, to have the unified bound, the total access time for a sequence $\mathcal{A}$ and any fixed finger f is $O(\sum_{i=1}^{m} \log(\min\{d_{\mathcal{A}}(e_i, f), m/h_{\mathcal{A}}(e_i), w_{\mathcal{A}}(i)\}))$.

There are implication relationships between the aforementioned distribution-sensitive properties. Iacono [11] observed that the working-set property implies the static-optimality and static-finger properties. We have recently proved the implication of the unified bound from the working-set bound [7], indicating that both bounds are asymptotically equivalent. That is, whereas Iacono [11] showed that the working-set property guarantees the minimum of the three bounds, we [7] showed that it even guarantees the minimum per operation of those bounds.

Distribution-sensitive data structures are not limited to search trees. Also priority queues have been designed and analyzed in the context of distribution-sensitivity [2,6,10,12]. In the comparison model, it is easy to observe that a priority queue with constant insertion time cannot have the sequential-access property and hence cannot as well have the dynamic-finger property; for otherwise, a sequence of insertions followed by a sequence of minimum-deletions would list the elements in sorted order in linear time (contradicting the $\Omega[n \log n)$ lower bound for comparison-based sorting). Alternatively, the working-set property has been of main interest for priority queues. Iacono [10] proved that pairing heaps [9] satisfy the working-set property as follows: in a heap of maximum size n, it takes $O(\log(\min\{o_x, n\}))$ amortized time to delete the minimum x, where o_x is the number of operations performed since x's insertion. Funnel-heaps [2] are I/O-efficient heaps for which it takes $O(\log(\min\{i_x, n\}))$ time to delete the minimum x, where i_x is the number of insertions performed since x's insertion. Elmasry [6] gave a priority queue supporting the deletion of the minimum x in $O(\log w_x)$ worst-case time, where w_x is the number of elements inserted after the insertion of x and are still present in the priority queue when x is deleted (note that $w_x \leqslant i_x \leqslant o_x$); we briefly review this priority queue in Section 2. None of the aforementioned priority queues supports *delete* (as introduced, they only support *delete-min*) within the working-set bound. In Section 3, we present a priority queue that supports *insert* in worst-case constant time, and *delete-min*, *access-min*, *delete*, and *decrease* in $O(\log w_x)$ worst-case time. For *delete* and *decrease*, we assume that a pointer to the designated node is readily given.

One natural sequence of operations is the first-in-first-out type. Data structures sensitive to these sequences must operate fast on elements that have been least-recently accessed. This distribution-sensitive property is referred to as the "queueish" property in [12]. In the context of priority queues, such property states that the time to delete the minimum x is $O(\log q_x)$, where q_x is the number of elements inserted before x and still present in the priority queue when x is deleted. Note that $q_x = n - w_x$, where n is the number of elements currently present in the priority queue. A priority queue with the queueish property is presented in [12]; this priority queue does not support *delete* either. It is also shown in [12] that no binary search tree can be sensitive to this property.

It remained open whether there exists a priority queue sensitive to both the working-set and the queueish properties. We resolve the question affirmatively by presenting such a priority queue in Section 4. In Section 5, we introduce the time-finger property. This property encapsulates the queueish and the working-set properties, and thus also captures the static-finger, static-optimality, and unified-bound properties. As these properties are the complete list of distribution-sensitive properties known for priority queues, we refer to the time-finger property as the "unifying property for priority queues". In consequence, we modify our priority queue to satisfy this unifying property.

We rely on the notion of *numeral systems* in our worst-case construction. A numeral system is a way for representing numbers with rules governing the operations performed on them. There is a connection between numeral systems and data-structural design [3,16]. The idea is to relate the number of objects of a specific type in the data structure to the value of a digit. The basic operations that we use are the increment and decrement of any digit. An *extended-regular binary number* [3] uses the digits $\{0, 1, 2, 3\}$ with the constraints that between any two 3's there is a digit other than 2, and between any two 0's there is a digit other than 1. The extended-regular numeral system allows for increments and decrements of arbitrary digits with a constant number of digit changes per operation (see [8,13] for the implementation details of this numeral system).

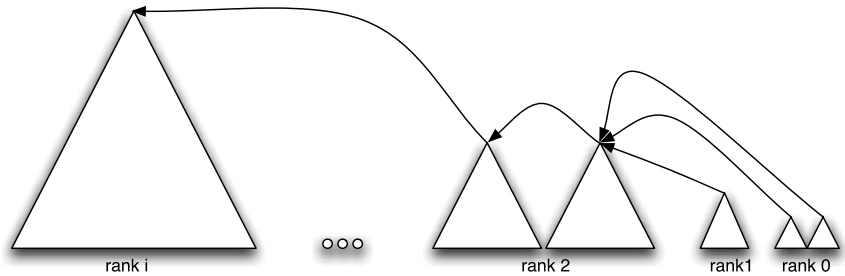

Fig. 1. $(2,3)$-binomial trees comprise the priority queue: The ranks of the trees are non-decreasing from right to left. Prefix-minimum pointers are maintained at the roots of the trees. Each tree root points to the minimum root to the left of it, including itself.

2. A priority queue with the working-set property

Our priority queue builds on the priority queue in [6], which supports *insert* in constant time and *delete-min* within the working-set bound. The advantage of the priority queue in [6] over those in [2,9,10] is that it satisfies the stronger working-set property, in which elements that are deleted do not count towards the working set. Next we outline the structure of this priority queue.

The priority queue in [6] is comprised of heap-ordered $(2,3)$-binomial trees. As defined in [6], the subtrees of the root of a $(2,3)$-binomial tree of rank r are $(2,3)$-binomial trees; there are one or two children having ranks $0,1,\ldots,r-1$, ordered in a non-decreasing rank value from right to left. A $(2,3)$-binomial tree of rank 0 is a single node. It is easy to verify that the rank of a $(2,3)$-binomial tree that contains n nodes is $\Theta(\log n)$. The ranks of the $(2,3)$-binomial trees of the priority queue are as well non-decreasing from right to left. For the amortized solution, there are at most two (possibly zero) trees per rank. For the worst-case solution, the number of trees per rank obeys a variation of the extended-regular numeral system [3], which guarantees that the ranks of any two adjacent trees differ by at most two. The root of every $(2,3)$-binomial tree has a pointer to the root with the minimum value among the roots to its left, including itself. Such *prefix-minimum* pointers allow for finding the overall minimum element in constant time, with the ability to maintain such pointers after deleting the minimum in time proportional to the rank of the deleted node. Note that the prefix-minimum pointer of a root can be updated in constant time by comparing the value stored at this root with the value referenced by the pointer at the root to its left (assuming that the pointer at this latter root is up to date). Fig. 1 illustrates the structure of this priority queue.

Two primitive operations are used: *split* and *join*. A $(2,3)$-binomial tree of rank r can be split into two or three trees of rank $r-1$; this is done by detaching the one or two children of the root that have the highest rank, i.e. with rank $r-1$. On the other hand, two or three $(2,3)$-binomial trees of rank $r-1$ can be joined to form a $(2,3)$-binomial tree of rank r; this is done by making the root(s) with the larger value the leftmost child(ren) of the one with the smallest value. Another special form of the join operation, that is needed for the construction, is to join a tree of rank $r-1$ and a tree of rank $r-2$. To accomplish this task, the first tree is split then all the resulting trees are joined: If there exist three trees of rank $r-2$ after the split, they are joined to get a tree of rank $r-1$. Otherwise, the split results in four trees of rank $r-2$; each pair is joined and then the resulting two trees are joined, ending with a tree of rank r.

A chronological total order is maintained according to the time the elements were inserted. If a binomial tree T_1 is to the right of another T_2, then all the elements in T_1 must have been inserted after those in T_2. Furthermore, within an individual tree, the target was that the preorder traversal of elements with a right-to-left precedence to subtrees would indicate the insertion order of these elements. However, when performing join operations this ordering is occasionally disobeyed by possibly reversing the order of two entire subtrees. To keep track of the correct ordering it is enough to maintain a *reverse bit* with every node x; such reverse bit indicates whether the elements in x's subtree were inserted before or after the element in x's parent plus those in the descendants of the right siblings of x. When a join is performed the corresponding reverse bit is properly set. When a split is performed the resulting trees are positioned according to the reverse bits of the detached children of the root.

With the join and split operations in hand, it is possible to detach the root of a $(2,3)$-binomial tree of rank r and reconstruct the tree again as a $(2,3)$-binomial tree with rank $r-1$ or r while maintaining the chronological order; this is done by repeated joins and splits starting from the rightmost subtrees of the deleted root to the leftmost (see [6] for the details).

To perform an *insert* operation, a new single node that holds the inserted element is added as the rightmost tree in the priority queue. This may give rise to repeated joins as long as there are three trees with the same rank; the number of such joins is amortized constant, resulting in a constant amortized cost per insertion. After performing the joins, the prefix-minimum pointer of the surviving root is updated. For the worst-case solution, the underlying extended-regular numeral system guarantees at most two joins per *insert* [3,8,13].

To perform a *delete-min* operation, the tree T of the minimum root is identified via the prefix-minimum pointer of the rightmost tree. The tree T is reconstructed as a $(2,3)$-binomial tree after detaching its root. This may be followed by a split

and a join if the rank of T is now one less than its original rank. Finally, the prefix-minimum pointers of the trees to the right of and including T are updated. For the amortized solution, several splits of T may follow: Let T' be the tree to the right of the tree T; starting with T, the rightmost tree resulting from previous splits is repeatedly split until the resulting trees of the split and T' have consecutive ranks. This splitting is unnecessary for the worst-case solution. It is not hard to conclude that the cost of the *delete-min* operation is $O(r)$, where r is the rank of the deleted node. There are $O(w_x)$ elements in the trees to the right of T, and hence the number of such trees is $O(\log w_x)$. For the worst-case solution, it then follows that the rank of the deleted node x is $O(\log w_x)$. For the amortized solution, an extra lemma [6, Theorem 1] establishes the same bound in the amortized sense.

3. Supporting *delete*, *access-min* and *decrease*

The existing distribution-sensitive priority queues [2,6,10,12] do not support *delete* within the working-set bound. In this section we modify the construction outlined in Section 2 to support *delete* within the working-set bound. Including *delete* in the repertoire of operations is not hard but should be done carefully. The main challenge is to maintain the elements in chronological order.

Let x be the node to be deleted. We traverse the ancestors of x upwards via the parent pointers until we reach the root of its tree. We use two stacks; a right stack and a left stack. Starting at the root, we repeatedly split the current subtree into two or three subtrees. While continuing to split the subtree that contains x, the other one or two subtrees are pushed onto the stacks, until we end up with a subtree whose root is x. (Depending on the reverse bit and the relative position of the root of a subtree to that containing x, we push the subtree on either the right or left stack.) At this stage, we delete x analogously to the *delete-min* operation: the node x is detached and the subtrees resulting from removing x are incrementally joined from right to left, while possibly performing one split before each join. Next we have to work our way up to the root and join all subtrees which we have introduced by splits on the way down from the root. The one or two subtrees that have the same rank are repeatedly popped from the stacks and joined with the current subtree, while possibly performing one split before each join. Once the two stacks are empty, a split and a join may be performed if the resulting tree has rank one less than its original rank.

The total order is correctly maintained by noting that the only operations employed are *split* and *join*, which are guaranteed to set and use the reverse bits correctly. Since the height of a $(2,3)$-binomial tree is one plus its rank, the time bound for *delete* is $O(r)$, where r is the rank of the tree that contains the deleted node. This establishes the same time bound as that for *delete-min*.

Using the prefix-minimum pointers, minimum finding is possible in constant time. However, the version of the operation that is considered an access cannot be supported in time asymptotically less than that of *delete-min*; otherwise, a following *delete-min* operation would have to be supported in constant time according to the working-set property. An *access-min* operation for an element x can be made to run in $O(\log w_x)$ time by executing a *delete-min* operation followed by a reinsertion.

Also the *decrease* operation that is considered an access cannot be performed in asymptotically less time than *delete*; otherwise, a *delete* operation can be performed by executing a *decrease* operation with decrease value of zero, which essentially brings the element to the front of the working set, followed by a *delete* operation that should now run in constant time. A *decrease* operation for an element x can be made to run in $O(\log w_x)$ time by executing a *delete* operation followed by a reinsertion.

Theorem 1. *The priority queue presented in this section supports insert in constant time, and delete-min, access-min, delete, and decrease of an element x in $O(\log w_x)$ time, where w_x is the number of elements accessed after the last access to x and still present in the priority queue at the current operation.*

4. Incorporating the queueish property

The queueish property for priority queues indicates that the time to perform an access operation (*delete-min*, *delete*, *access-min*, or *decrease*) to an element x is $O(\log(n - w_x))$, where n is the number of elements currently present in the priority queue and w_x is the number of elements accessed after the last access to x and still present in the priority queue. In other words, the queueish property states that the time to perform one of the aforementioned operations on an element x is $O(\log q_x)$, where $q_x = n - w_x$ is the number of elements last accessed prior to x and still present in the priority queue. Queaps [12] are queueish priority queues that support *insert* in amortized constant time and support any of the other operations on x in amortized $O(\log q_x)$ time.

We extend our priority queue from Section 3, in addition to supporting the working-set bound, to also support the operations within the queueish bound. Instead of having the ranks of the trees of the queue non-decreasing from right to left, we divide the queue into two sides, a right queue and a left queue. The ranks of the trees of the right queue are monotonically non-decreasing from right to left, and those of the left queue are monotonically non-decreasing from left to right. We also impose the constraint that the difference in rank between the largest tree on each side is at most one. Fig. 2 depicts the new priority queue.

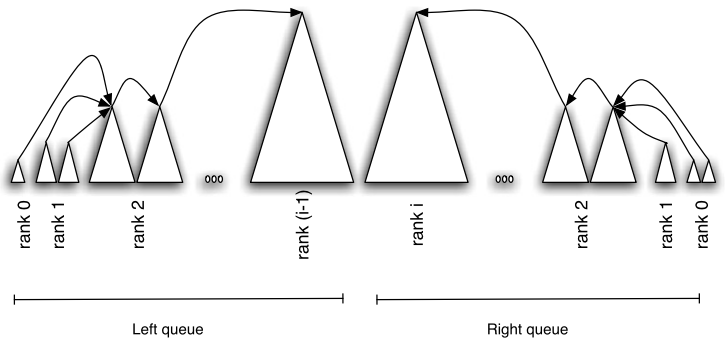

Fig. 2. A priority queue satisfying the queueish property: the priority queue is comprised of two queues one of which has tree ranks increasing from right to left (right queue) and one increasing from left to right (left queue). The ranks of the largest trees on the two sides must differ by at most one. The prefix-minimum pointers are separate for each side.

The prefix-minimum pointers in the left and right queues are kept independently. In the right queue, the root of each tree maintains a pointer to the root with the minimum value among those in the right queue to the left of it. Conversely, in the left queue, the root of each tree maintains a pointer to the root with the minimum value among those in the left queue to the right of it. To find the overall minimum value, both the left and right queues are probed.

Insertions are performed exactly as before in the right queue. The *delete-min* operation is performed in the left or right queue depending on where the minimum lies. The other operations are performed as we have mentioned earlier.

We must maintain the invariant that the difference in rank between the largest tree on the left and right sides is at most one. Since the chronological order is maintained among our trees, this invariant guarantees that the rank of the tree of an element x is $O(\log(\min\{w_x, q_x\}))$. As a result of an insertion or a deletion, this difference in ranks may become two. Once the highest rank on one side is two more than that on the other side, we need to move some trees between the two sides. Assume that the highest rank on side "A" is r and that on side "B" is $r-2$. To maintain the invariant, we do the following:

Case 1. If there are two trees of rank r on side "A", move the appropriate (to maintain the chronological order) one of the two to side "B".

Case 2. Otherwise, split the only tree of rank r on side "A" into two or three trees of rank $r-1$. We now have $k \in [2..5]$ trees of rank $r-1$ on side "A".
- (a) If $2 \leqslant k \leqslant 4$, move the appropriate $\lfloor k/2 \rfloor$ trees to side "B", and stop.
- (b) Otherwise ($k = 5$), move the appropriate two trees to side "B", and then join the appropriate two trees among the remaining three on side "A" to form a tree of rank r.

For the worst-case solution, using the above procedure, the ranks of the trees on each side would still obey the extended-regular numeral system.

Once trees are moved from one side to the other, all the prefix-minimum pointers on both sides of the priority queue may need to be updated. However, for the amortized solution, the next lemma indicates that this work does not affect the claimed amortized bounds per operation.

Lemma 1. *Consider the two-sided priority queue presented in this section. For the amortized solution, updating the prefix-minimum pointers only accounts for an extra constant factor in the amortized cost per operation.*

Proof. Consider the moment just after moving trees between the two sides of the priority queue and updating the prefix-minimum pointers. For this action to happen again, the highest ranks on one or both sides have to change. One possibility is that such a rank on one side decreases as a result of a *delete-min* or *delete* operation. In this case, the deletion would take $\Theta(\log n)$ time; there is surely enough time to update the prefix-minimum pointers within the same asymptotic bound, if necessary. The other possibility is that the highest rank on one side increases as a result of *insert* operations. We show next that, in such a case, updating the pointers is done only after many *insert* operations.

For example, lets analyze Case 2(a). After the pointer updates, the highest rank on both sides is $r-1$. The highest rank can become r fairly soon, even after one insertion, but this would leave no trees with any of the ranks smaller than r. In other words, this side of the priority queue would be composed of only one tree the rank of which is r. For the highest rank to become $r+1$, there should be enough inserted elements to produce two more trees of rank r. Since the size of each such tree is at least 2^r, at least 2^{r+1} more *insert* operations are to be performed before the next pointer updates. Since the number of pointer updates is $O(r)$ and each update requires a constant amount of work, the constant amortized cost per *insert* easily covers those costs. The other cases are quite similar to this case, and the same arguments apply. □

To guarantee the costs in the worst case, updating those prefix-minimum pointers is to be done incrementally with the upcoming insertions as follows. With every *insert* operation we update a constant number of the prefix-minimum pointers; this updating process starts from the root with the highest rank to that with the lowest rank on each side. As a consequence, at any point of time, it is possible that the roots with the higher ranks have their prefix-minimum pointers up to date while the other roots do not. A reference is maintained to the last root x whose prefix-minimum pointer is up to date. In accordance, for a *delete-min* operation to find the minimum on one side of the priority queue it consults the prefix-minimum pointer of the root with the lowest rank as well as that of the root x. The preceding lemma shows that there will be enough insertions to finish this updating process before a new one needs to be initiated.

A deletion of a node x in a tree of rank r would still cost $O(r)$ time, but now $r = O(\log(\min\{w_x, q_x\}))$ in the amortized sense (for the amortized solution), or in the worst-case sense (for the worst-case solution).

Theorem 2. *The two-sided priority queue presented in this section supports insert in constant time, and delete-min, access-min, delete, and decrease of an element x in $O(\log(\min\{w_x, q_x\}))$ time, where w_x and q_x are the number of elements accessed after, respectively before, the last access to x and still present in the priority queue at the current operation.*

5. Supporting multiple time fingers

We define time fingers $t_1, t_2, \ldots, t_c$ as time instances within the sequence of operations, which are freely set by the implementer. We define the working-set of an element x with respect to time finger t_i, $w_x(t_i)$, as the number of elements that have been last accessed in the window of time between the last access to x and t_i and are still present in the priority queue. We say that a priority queue satisfies the multiple-time-finger property if the time to access x is $O(c + \log(\min_{i=1}^{c}\{w_x(t_i)\}))$. It is not hard to see that the working-set property is equivalent to having a single time finger $t_1 = +\infty$, and the queueish property is equivalent to having a single time finger $t_1 = 0$. The priority queue presented in Section 4, which supports both the working-set and the queueish properties, has two time fingers $t_1 = 0$, $t_2 = +\infty$. In this section, we present a priority queue that satisfies the property for any number of time fingers.

The structure consists of multiple two-sided priority queues, as those designed in Section 4. We start with a single two-sided priority queue PQ_0, and at each point when a new time finger is introduced we finalize the priority queue and start a new one. Therefore, corresponding to c time-fingers $t_1 = 0, \ldots, t_c = \infty$, we have $c - 1$ two-sided priority queues $PQ_1, \ldots, PQ_{c-1}$.

Insertions are performed in the last (at the time when the insertion is performed) priority queue, and take constant time each (by Theorem 2). For *delete* operations, we are given a reference to an element x to delete. We determine to which priority queue PQ_j the element belongs and delete it. This requires $O(\log(\min\{w_x(t_j), w_x(t_{j+1})\}))$ time (by Theorem 2). Since x belongs to PQ_j, for any $i < j$, $w_x(t_j) \leqslant w_x(t_i)$, and for any $i > j + 1$, $w_x(t_{j+1}) \leqslant w_x(t_i)$. It follows that $\log(\min\{w_x(t_j), w_x(t_{j+1})\}) = \log(\min_{i=1}^{c}\{w_x(t_i)\})$. For the *delete-min* operation, it suffices to note that finding the minimum element per queue takes constant time. Therefore, we can determine in $O(c)$ time the priority queue containing the minimum and perform the operation there. The running-time argument is the same as that for the *delete* operation.

Theorem 3. *Given time fingers $t_1, t_2, \ldots, t_c$, the priority queue presented in this section supports insert in constant time, and the operations delete-min, access-min, delete, and decrease of an element x in $O(c + \log(\min_{i=1}^{c}\{w_x(t_i)\}))$ time, where $w_x(t_i)$ is the number of elements that have been accessed in the window of time between the last access to x and t_i and are still present in the priority queue at the current operation.*

6. Summary

Our focus was on distribution-sensitive priority queues. Provably, priority queues cannot satisfy the sequential-access property and in accordance neither the dynamic-finger nor the unified conjecture. We therefore considered other distribution-sensitive properties, namely: the working-set and the queueish properties; we presented a priority queue that satisfies both properties. Our priority queue builds on the priority queue of [6], which supports *insert* in constant time and *delete-min* in the working-set time bound. We showed that the same structure can also support *delete* within the working-set bound. We then modified the structure to satisfy the queueish property as well. It is worth noting that our priority queue supports the stronger definition of the working-set and the queueish properties, in which the elements deleted are not accounted for in the time bounds. Our recent result about the asymptotic equivalence of the working-set bound and the unified bound [7] implies that our priority queue also satisfies the static-optimality, static-finger, and unified bounds. We defined the notion of time fingers, which encapsulates the working-set and the queueish properties. The priority queue described thus far has two time fingers. We generalized our priority queue to possibly support any number of time fingers.

The bounds mentioned are amortized. However, we showed that the time bounds for the working-set and queueish properties can also be made to work in the worst case. More generally, the multiple-time-finger bounds can be made to work in the worst case. However, the time bounds for the other properties — static optimality, static finger, and unified bound — naturally remain amortized.

References

[1] M. Badoiu, R. Cole, E.D. Demaine, J. Iacono, A unified access bound on comparison-based dynamic dictionaries, Theoretical Computer Science 382 (2007) 86–96.
[2] G.S. Brodal, R. Fagerberg, Funnel heap – a cache oblivious priority queue, in: Proceedings of the 13th International Symposium on Algorithms and Computation, in: Lecture Notes in Computer Science, vol. 2518, Springer-Verlag, 2002, pp. 219–228.
[3] M.J. Clancy, D.E. Knuth, A programming and problem-solving seminar, Technical Report STAN-CS-77-606, Department of Computer Science, Stanford University, 1977.
[4] R. Cole, On the dynamic finger conjecture for splay trees. Part II: Finger searching, SIAM Journal on Computing 30 (2000) 44–85.
[5] A. Elmasry, On the sequential access theorem and dequeue conjecture for splay trees, Theoretical Computer Science 314 (2004) 459–466.
[6] A. Elmasry, A priority queue with the working-set property, International Journal of Foundations of Computer Science 17 (2006) 1455–1466.
[7] A. Elmasry, A. Farzan, J. Iacono, A unifying bound for distribution-sensitive priority queues, in: Proceedings of the 22nd International Workshop on Combinatorial Algorithms, in: Lecture Notes in Computer Science, vol. 7056, Springer-Verlag, 2011, pp. 209–222.
[8] A. Elmasry, J. Katajainen, Worst-case optimal priority queues via extended regular counters, in: Proceedings of the 7th International Computer Science Symposium in Russia, in: Lecture Notes in Computer Science, vol. 7353, Springer-Verlag, 2012, pp. 130–142.
[9] M.L. Fredman, R. Sedgewick, D.D. Sleator, R.E. Tarjan, The pairing heap: a new form of self-adjusting heap, Algorithmica 1 (1986) 111–129.
[10] J. Iacono, Improved upper bounds for pairing heaps, in: Proceedings of the 7th Scandinavian Workshop on Algorithm Theory, in: Lecture Notes in Computer Science, vol. 1851, Springer-Verlag, 2000, pp. 32–45.
[11] J. Iacono, Distribution-sensitive data structures, Ph.D. thesis, Rutgers, The state University of New, Jersey, New Brunswick, New Jersey, 2001.
[12] J. Iacono, S. Langerman, Queaps, Algorithmica 42 (2005) 49–56.
[13] H. Kaplan, N. Shafrir, R.E. Tarjan, Meldable heaps and Boolean union-find, in: Proceedings of the 34th Annual ACM Symposium on Theory of Computing, ACM, 2002, pp. 573–582.
[14] D.D. Sleator, R.E. Tarjan, Self-adjusting binary search trees, Journal of the ACM 32 (1985) 652–686.
[15] R.E. Tarjan, Sequential access in splay trees takes linear time, Combinatorica 5 (1985) 367–378.
[16] J. Vuillemin, A data structure for manipulating priority queues, Communications of the ACM 21 (1978) 309–315.
[17] H.E. Williams, J. Zobel, S. Heinz, Self-adjusting trees in practice for large text collections, Software—Practice and Experience 31 (2001) 925–939.